PROGETTO ARCHITETTURE E PROGRAMMAZIONE DEI SISTEMI DI ELABORAZIONE

ANNO ACCADEMICO

2021/2022

STUDENTI:

Amirato Simone 235520

D’Atri Fulvio 235344

De Luca Francesco 235300

Sommario

1. Introduzione

Lo svolgimento della consegna prevede la realizzazione di un algoritmo in grado di simulare il comportamento di un banco di pesci. Questo algoritmo prende il nome di *Fish School Search* (FSS) ed è un algoritmo di ottimizzazione, ispirato, come detto precedentemente, al comportamento dei banchi di pesci e dei meccanismi di alimentazione degli stessi. In particolare, i pesci nuotano verso il "gradiente positivo" per poter mangiare e prendere così peso. Dato che i pesci più pesanti hanno maggiore influenza sul banco nel processo di ricerca collettiva di cibo, il baricentro del banco tende a spostarsi verso i posti migliori dello spazio di ricerca. Nello specifico, sono state realizzate più versioni. Si è partiti implementando in linguaggio *C* l'algoritmo, il quale elabora dati sia a 32 bit (*float*) che a 64 bit (*double*); successivamente, alcuni tra i metodi utilizzati nella versione in *C* sono stati implementati in linguaggio *Assembly* con l'utilizzo del repertorio *SSE* (*Streaming SIMD Extensions*) per la versione a 32 bit e *AVX* (*Advanced Vector Extensions*) per la versione a 64 bit. Più in particolare, come richiesto dalla traccia, l'obiettivo era adottare delle tecniche di ottimizzazione dell'hardware affinché i tempi di esecuzione dell'algoritmo diminuiscano. In generale le tecniche trattate nel corso sono 4:

- *SIMD-Based* (*Single Instruction, Multiple Data*): con questo tipo di tecniche è possibile eseguire operazioni su più dati con una singola istruzione, grazie alle unità funzionali che operano sui vettori (ad esempio i registri XMM e YMM nei repertori SSE e AVX, rispettivamente). La tecnica più usata di questa categoria è la *Loop Vectorization*, che garantisce, all'interno dei cicli, un fattore di parallelismo pari alla dimensione del vettore.
- *ILP-Based* (*Instruction Level Parallelism*): è l'insieme di tecniche che stimolano il parallelismo implicito, ovvero quel parallelismo che gestisce il processore indipendentemente dalle istruzioni del programmatore. Un esempio di tecnica ILP-Based è la *Loop Unrolling* , che consente di ridurre i costi derivati dalle istruzioni di gestione del ciclo (aggiornamento, controllo condizione, salti condizionati), che vengono dunque distribuiti sulle istruzioni srotolate, le quali con molta probabilità sono parallelizzate implicitamente dal processore; ciò è possibile solo con operazioni indipendenti tra loro.
- *CACHE-Based*: sono delle tecniche aventi lo scopo di minimizzare il numero di accessi in memoria e quindi ridurre il numero di *cache-miss*. La tecnica più usata è denominata *Loop Blocking* che consiste nel raggruppare i dati in un insieme di blocchi e quindi di iterare non più sull'intera struttura dati ma solo su suddetti blocchi. Date le dimensioni ridotte che questi presentano è possibile spostarli interamente in cache, così da ridurre il numero degli accessi in memoria.
- *MIMD-Based* (*Multiple Istruction, Multiple Data*): sono tecniche che sfruttano la presenza di unità centrali duplicate. Un modo per usare le tecniche MIMD nel linguaggio C è l'utilizzo delle direttive *OpenMP*, queste vengono poste sopra l'intestazione dei cicli che si vogliono parallelizzare; in particolare, la CPU cercherà di istanziare un thread per ogni iterazione del ciclo. Chiaramente, anche in questo caso, ciò è possibile solo se le operazioni sono indipendenti tra loro.

La relazione seguirà il seguente schema: nella prima parte sarà descritta come verrà salvata in memoria la matrice *x* (la matrice che descrive la posizione di ogni pesce) e le altre matrici usate nell'algoritmo e verranno illustrate le motivazioni di tale scelte. …

Si è partiti realizzando la prima versione del progetto in codice C. Già in questa prima fase sono state fatte delle importanti scelte progettuali, sulla base di alcuni test effettuati. Nello specifico è stata

implementata una versione che lavorasse su dati memorizzati per righe e una versione per colonne. Al completamento di queste versioni sono stati effettuati dei test per valutare quale presentasse prestazioni migliori; questi sono stati a favore della versione per righe. Scelta la rappresentazione di riferimento sono state realizzate le prime versioni in Assembly. La prima ottimizzazione effettuata è stata quella di tradurre la quasi totalità dei metodi da C ad Assembly usando il repertorio SSE su architettura a 32 Bit; in particolare sono state adottate le istruzioni a singoli *float* (*MOVSS*). Raggiunta una versione funzionante, è stata effettuata un'ulteriore ottimizzazione, mediante la tecnica Loop Vectorization. Successivamente, si è fatto uso dell'ultima tecnica di ottimizzazione per quanto riguarda la versione a 32 Bit, ovvero la Loop Unrolling. Dopo alcune riflessioni, si è arrivati a conclusione che per la struttura che il codice presenta, il cache blocking non fornisce il livello di ottimizzazione atteso, come verrà spiegato più avanti.

Raggiunta la versione finale del codice basato su architettura a 32 Bit, si è passati ad implementare le stesse tecniche di ottimizzazione hardware, realizzate in precedenza, per l'architettura a 64 Bit usando il repertorio AVX.

Infine, per le due versioni, 32/SSE e 64/AVX, sono state utilizzate le direttive OpenMP al fine di fornire un'ottimizzazione di tipo MIMD.

Nei capitoli successivi sarà anche mostrato come il codice cambia in base alle diverse ottimizzazioni hardware implementate; il metodo usato come riferimento sarà il metodo •*type func(params* input, MATRIX x, int i)*.

2. Versione C:

Come suggerito dalla traccia lo sviluppo dell'algoritmo FSS è stato scomposto in più metodi:

- *void inizializzazionePesi(params* input,VECTOR pesi, type* pesoTotale)*: il metodo inizializza le posizioni di ogni pesce con d valori casuali ed assegna ad ognuno un peso iniziale di $W_i/2$.
- *type func(params* input, MATRIX x, int i): func* restituisce il valore della funzione $f(x) = e^x + x^2 - c*x$ accedendo al vettore delle posizioni del pesce i, memorizzato nella matrice *x*.
- *void movimentoIndividuale(params* input, MATRIX deltaX, VECTOR deltaF, int* indRandom, VECTOR y)*: il metodo simula per ogni pesce uno spostamento individuale. Le nuove posizione generate vengono memorizzate in un vettore y_i. Per ognuna delle *d* posizione viene eseguita questa operazione:

$$\mathbf{y}_i(j) = \mathbf{x}_i(j) + rand(-1, 1) \cdot step_{ind}, j \in [1, d],$$

- I valori random sono presi da una struttura dati; la struttura contiene solo numeri tra 0 ed 1, e per fare in modo che il range vada da -1 ad 1 ogni numero viene raddoppiato e decrementato di 1. Inoltre, vengono calcolate le variazioni Δf_i *e* Δx_i. La prima viene calcolata come la differenza tra *func* calcolata sul vettore *y* per il pesce *i* e *func* calcolata sul vettore *x* per il pesce *i* salvando il risultato in un vettore chiamato *deltaF*, mentre la seconda esegue una sottrazione vettoriale tra *y* e *x*, successivamente salvata in un vettore $deltaX_i$.
- *void minimo(VECTOR deltaF, int n, type* min)*: il metodo calcola e restituisce la minima variazione di *deltaF* tra tutti i pesci.
- *void operatoreAlimentazione(params* input, VECTOR pesi, VECTOR deltaF)*: il metodo esegue l'operazione di alimentazione per ogni pesce secondo la formula. Vengono eseguiti dei controlli per verificare che la minima variazione di deltaF sia diversa da 0. La verifica è necessaria perché nel caso in cui nessun pesce trovi una posizione migliore *deltaF* sarà 0 per ogni pesce.
- *void calcolaI(VECTOR deltaF, MATRIX deltaX, params* input, VECTOR I)*: il metodo calcola il vettore I di d dimensioni secondo la formula:

$$\boldsymbol{I} = \frac{\sum_{i=1}^{n} \Delta \mathbf{x}_i \cdot \Delta f_i}{\sum_{i=1}^{n} \Delta f_i},$$

- *void movimentoIstintivo(params* input, MATRIX deltaX, VECTOR deltaF, VECTOR I):* dopo aver chiamato calcolaI il metodo procede ad eseguite la seguente operazione per ogni pesce:

$$\mathbf{x}_i = \mathbf{x}_i + \boldsymbol{I}$$

- *void calcolaBaricentro(VECTOR baricentro, type* pesoTotale, VECTOR pesi, params* input)*: il metodo calcola il vettore baricentro di d dimensioni secondo la formula:

$$\mathbf{B} = \frac{\sum_{i=1}^{N} \mathbf{x}_i \cdot W_i}{\sum_{i=1}^{N} W_i}$$

- *void calcolaDistanzaEuclidea(type* distanzaEuclidea, params* input, VECTOR baricentro, int i)*: il metodo restituisce la distanza euclidea tra il baricentro del banco di pesci ed il pesce in posizione i.
- *void movimentoVolitivo(params* input, int* indRandom, VECTOR baricentro, type* pesoTotale, type* nuovoPesoTotale)*: il metodo simula per ogni pesce lo spostamento volitivo applicando la seguente formula:

$$x_i = x_i \pm step_{vol} \cdot rand(0, 1) \cdot \frac{\mathbf{x}_i - \mathbf{B}}{dist(\mathbf{x}_i, \mathbf{B})},$$

In particolare, se il peso totale del banco è aumentato rispetto al valore precedentemente assunto, i pesci sono attratti verso il baricentro (segno − nell'equazione), altrimenti si allontanano (segno + nell'equazione). Il metodo richiama *calcolaDistanzaEuclidea.*

- void aggiornamentoParametri(params* input, type stepindInitial, type stepvol*Inital)*: il metodo si occupa di aggiorna ad ogni iterazione i parametri stepVol e stepInd come segue

$$step_{ind} = step_{ind} - \frac{step_{ind}(initial)}{It_{\max}},$$
$$step_{vol} = step_{vol} - \frac{step_{vol}(initial)}{It_{\max}}$$

- *void trovaOttimo(params* input)*: il metodo calcola il valore di *func* per ogni pesce e dopo aver trovato quello per cui il valore è minimo, sostituisce in *xh* il suo vettore delle posizioni.

I metodi vengono richiamati nel blocco delle iterazioni nel seguente ordine:

```
inizializzazionePesi(input, pesi, &pesoTotale);

for(it=0; it<input->iter; it++){
        movimentoIndividuale(input, deltaX, deltaF, &indRandom, y);
        operatoreAlimentazione(input, pesi, deltaF);
        movimentoIstintivo(input, deltaX, deltaF, I);
        calcolaBaricentro(baricentro, &nuovoPesoTotale, pesi, input);
        movimentoVolitivo(input, &indRandom, baricentro, &pesoTotale, &nuovoPesoTotale)
        aggiornamentoParametri(input, stepindInital, stepvolInital);
}

trovaOttimo(input);
```

Blocco iterazioni metodo fss

Ecco di seguito illustrato il codice del metodo di riferimento (*func*), scritto in linguaggio *C*. L'algoritmo come si può vedere ha il compito di calcolare il valore di funzione di un particolare pesce passato come parametro (*int i*). Esso è composto da un ciclo *for*, che itera sulle dimensioni della matrice *x* e sugli elementi del vettore dei coefficienti (*c*). Ad ogni iterazione vengono aggiornati due contatori: *esponente* e *cx*. Il primo accumula il quadrato del valore della posizione corrente (*x[d*i+j]*), mentre il secondo accumula il prodotto tra il valore della posizione corrente (*x[d*i+j]*) per il valore del coefficiente corrente (*input->c[j]*). Infine viene restituito la differenza tra *esponente* e *cx* sommati al valore della funzione esponenziale *exp(esponente)*.

```
type func(params* input, MATRIX x, int i){
        int j;
        type esponente = 0;
        type cx = 0;
        int d = input->d;
        for(j=0; j<d; j++){
                esponente += x[d*i+j]*x[d*i+j];
                cx += input->c[j]*x[d*i+j];
        }
        return exp(esponente) + esponente - cx;
}
```

Metodo func scritta in linguaggio C

2.1. Rappresentazione dei dati in memoria:

La prima scelta progettuale ha riguardato il tipo di rappresentazione dei dati in memoria. Al fine di avere risultati concreti su cui basare la scelta, sono state realizzate le due versioni in C, di cui prima. È stato necessario ordinare in maniera opportuna i cicli di iterazione in base alla disposizione dei dati.

```
for(i=0; i<n; i++){
        *pesoTotale = *pesoTotale + pesi[i];
}
for(j=0; j<d; j++){
        baricentro[j] = 0;
        for(i=0; i<n; i++){
                baricentro[j] += x[i+j*n] * pesi[i];
        }
        baricentro[j] = baricentro[j] / (*pesoTotale);
}
```

Rappresentazione per colonne

```
for(j=0; j<d; j++)
        baricentro[j]=0;
for(i=0; i<n; i++){
        *pesoTotale = *pesoTotale + pesi[i];
        for(j=0; j<d; j++){
                baricentro[j] += x[d*i+j] * pesi[i];
        }
}
for(j=0; j<d; j++)
        baricentro[j] = baricentro[j] / (*pesoTotale);
```

Rappresentazione per righe

Nelle due figure è rappresentato il metodo *calcolaBaricentro* nelle due differenti rappresentazioni della matrice delle posizioni dei pesci (*x*). A sinistra vi è la rappresentazione per colonne, infatti, come si può notare le iterazioni sono predisposte prima per colonna e poi per riga, mentre a destra vi è la rappresentazione per righe e le iterazioni sono invertite rispetto al caso precedente.

L'idea di disporre le iterazioni ordinate in base al tipo di rappresentazione dei dati in memoria è guidata da due fattori: ridurre il numero degli accessi in memoria sfruttando la cache e agevolare l'utilizzo di istruzioni vettorizzate nel linguaggio macchina.

Il metodo *calcolaBaricentro* ed il metodo *calcolaI* sono gli unici metodi avente un ordinamento delle iterazioni prima per colonne e poi per righe, favorendo la rappresentazione dei dati per colonne. I restanti metodi dell'algoritmo prevedevano un ordinamento naturale delle iterazioni prima per righe e poi per colonne, favorendo la rappresentazione dei dati per righe.

La strada percorsa è stata quella di rappresentare i dati in memoria per righe (da questo punto in avanti è dato per assunto che la rappresentazione adottata è quella per righe), questa scelta è dovuta alle seguenti motivazioni:

- La maggior parte dei metodi presentavano una disposizione delle iterazioni ordinate prima per righe e poi per colonne, portando i benefici descritti in precedenza.
- Sono state effettuate delle simulazioni per entrambi i tipi di rappresentazione e si è notato che i tempi di esecuzione della rappresentazione per righe sono risultati minori di quella per colonne, come si può vedere nella figura sottostante.

```
Coefficient file name: 'data/coeff32_256.ds2'
Random numbers file name: 'data/rand32_256_768_1250.ds2'
Initial fish position file name: 'data/x32_256.ds2'
Dimensions: 256
Number of fishes [np]: 768
Individual step [si]: 0.003000
Volitive step [sv]: 0.000400
Weight scale [w]: 0.040000
Number of iterations [it]: 1250
FSS time = 10.516 secs
FUNC=0.999840

Done.
```

Simulazione matrice salvata per righe

```
Coefficient file name: 'data/coeff32_256.ds2'
Random numbers file name: 'data/rand32_256_768_1250.ds2'
Initial fish position file name: 'data/x32_256.ds2'
Dimensions: 256
Number of fishes [np]: 768
Individual step [si]: 0.003000
Volitive step [sv]: 0.000400
Weight scale [w]: 0.040000
Number of iterations [it]: 1250
FSS time = 11.562 secs
FUNC=0.999840

Done.
```

Simulazione matrice salvata per colonne

2.2. Confronto tra le soluzioni:

Completata la stesura dell'algoritmo in linguaggio C, si è passati all'esecuzione, che ha evidenziato come l'errore tra il valore della soluzione calcolata e la soluzione ottima sia trascurabile; inoltre, all'aumentare del numero di iterazioni l'errore si avvicina sempre di più a zero. Nella figura sottostante sono rappresentate la soluzione dell'algoritmo in C e la soluzione del file *fss_sol.py*, che con l'ausilio della libreria *numpy*, calcola la soluzione ottima. La simulazione è stata effettuata con un numero di dimensioni pari a 8, un numero di pesci pari a 64 ed un numero di iterazioni pari a 30000. Come si può notare, i valori della posizione ottima calcolata rispetto a quella reale si discostano di valori inferiore al millesimo.

```
PARAMETERS:
        nx: 64 d: 8 iters: 30000
FILE SIZE f: 0.06437304615974426 G
FILE SIZE d: 0.12874609231948853 G
Genero il file rand.
Genero il file coeff.
Genero il file x init.
[-0.0202003  -0.01093588 -0.00033394 -0.02082672 -0.00534773 -0.01103847
 -0.01440705 -0.0218357 ]
```

Soluzione ottima calcolata da fss_sol.py (8-64-250)

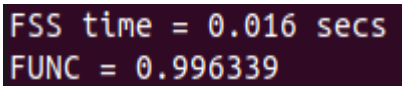
```
Coefficient file name: 'data/coeff32_8.ds2'
Random numbers file name: 'data/rand32_8_64_30000.ds2'
Initial fish position file name: 'data/x32_8.ds2'
Dimensions: 8
Number of fishes [np]: 64
Individual step [si]: 1.000000
Volitive step [sv]: 0.100000
Weight scale [w]: 10.000000
Number of iterations [it]: 30000
FSS time = 1.012 secs
xh: [-0.202005,-0.1093598,-0.00033498,-0.02082698,-0.0053498,-0.0110066,
-0.0146291,-0.0218877]

Done.
```

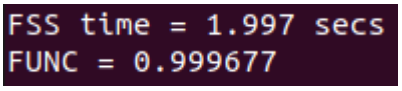
Soluzione ottima calcolata dall'algoritmo (8-64-250)

2.3. Tempi di esecuzione:

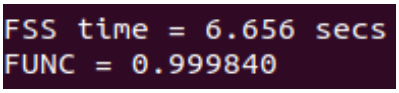
La versione dell'algoritmo scritto in linguaggio C è quella più basilare dal punto di vista delle prestazioni sui tempi di esecuzione, siccome non vi sono alcune ottimizzazioni hardware implementate (se non la scelta strategica della rappresentazione dei dati per righe). In figura sono rappresentati i tempi e il valore della funzione delle tre diverse simulazioni (8-64-250, 125-735-750, 256-768-1250), mettendo in evidenza i tempi di esecuzione dell'algoritmo senza ottimizzazioni hardware. Ovviamente come è auspicabile si nota un aumento dei tempi di esecuzione all'aumentare della matrice delle dimensioni dei pesci e del numero di iterazioni.



(8-64-250)



(125-735-750)



(256-768-1250)

3. Versione 32 bit/SSE:

La prima versione realizzata è quella a 32 Bit; in particolare, dopo aver scritto l'algoritmo in linguaggio C si è cercato di ottimizzare il più possibile, con l'utilizzo del repertorio SSE di Assembly e anche con le tecniche MIMD, come già anticipato. Nello specifico le tecniche usate nell'ambito del repertorio SSE sono state Loop Vectorization e Loop Unrolling; invece, per quanto riguarda le tecniche MIMD, queste sono state adoperate mediante le direttive OpenMP. Più avanti verranno mostrati con maggiore dettaglio i benefici portati sui tempi di esecuzione da queste tecniche di ottimizzazione.

3.1. Assembly con istruzioni scalari:

Come accennato nell'introduzione, la prima ottimizzazione realizzata è stata quella di trascrivere i metodi implementati precedentemente in linguaggio C, in linguaggio macchina *ASSEMBLY*, adottando il repertorio di istruzioni *SSE* come richiesto dalla traccia del progetto. In particolare, si è fatto uso delle istruzioni a singolo float (istruzioni scalare e non vettorizzate, esempio *MOVSS*). In figura sono mostrate le chiamate dei metodi scritti in *ASSEMBLY*, ovvero avente la dicitura *extern* in modo da far capire al compilatore *gcc* che il corpo delle funzioni dichiarate si trovi in un file *.nasm* esterno.

```
extern void func32(params* input, MATRIX xy, int i, type* esponente, type* ris);

extern void movimentoIndividuale1(params* input, int* indRandom, MATRIX y,int zeri);

extern void movimentoIndividuale2(params* input, MATRIX y,MATRIX deltaX, VECTOR deltaF);

extern void operatoreAlimentazione(params* input, VECTOR pesi, VECTOR deltaF, type min);

extern void movimentoIstintivo(params* input, VECTOR I);

extern void calcolaI(params* input, VECTOR I, VECTOR deltaF, MATRIX deltaX);

extern void calcolaBaricentro(VECTOR baricentro, type* pesoTotale, VECTOR pesi, params* input);

extern void movimentoVolitivo(params* input, int* indRandom, VECTOR baricentro, type* pesoTotale, type* nuovoPesoTotale);
```

Metodi extern

In figura sono rappresentati prima il codice di *func* in *C*, e poi quello in *ASSEMBLY* base. L'idea è stata quella di effettuare le operazioni viste in precedenza del metodo *func* in *C* direttamente in linguaggio macchina, mentre applicare la funzione esponenziale richiamando *exp()* dalla libreria *Math.h* di *C*. Nel codice *ASSEMBLY* si può vedere come le istruzioni sui registri *XMM* presentano la "s", indicando appunto che le istruzioni operano su singoli *float* e non su vettori di *float*. Da notare inoltre la parte finale, in cui vengono restituiti i valori di *esponente* e di *cx* (ovvero saranno scritti in memoria, nella locazione puntata dai rispettivi puntatori passati come parametro del metodo).

```
type func(params* input, MATRIX xy, int i){
        type esponente;
        type ris;
        func32(input, xy, i, &esponente, &ris);
        return ris + exp(esponente);
}
```

Metodo func in C che richiama func32

```
mov     EAX, [EBP+input]
mov     ECX, [EBP+xy]
mov     ESI, [EAX + c]
xorps   XMM0, XMM0
xorps   XMM2, XMM2
mov     EDX, 0
forjf:
imul    EDI, [EAX+d], dim
imul    EDI, [EBP+i]

add     EDI, EDX
movss   XMM1, [ECX + EDI]
mulss   XMM1, XMM1
addss   XMM0, XMM1
movss   XMM1, [ECX + EDI]
mulss   XMM1, [ESI + EDX]
addss   XMM2, XMM1

add     EDX, dim
mov     EDI, [EAX+d]
imul    EBX, EDI, dim
cmp     EDX, EBX
jb      forjf

mov     EBX, [EBP+indExp]
movss   [EBX], XMM0

subss   XMM0, XMM2
mov     EAX, [EBP + indRis]
movss   [EAX], XMM0
```

Metodo func32 ASSEMBLY versione base

Solamente questo primo approccio, ovvero quello di scrivere il codice dei metodi direttamente in linguaggio macchina e non lasciare che questo venga fatto dal compilatore, ha portato i primi miglioramenti in termini di efficienza, come si evince dai tempi di esecuzione che si hanno eseguendo con i valori al massimo (256-768-1250). Come si vede in figura è presente un vantaggio di circa 5 secondi nella versione in *ASSEMBLY* rispetto la versione in *C*.

```
FSS time = 6.656 secs
FUNC = 0.999840
```

Versione C

```
FSS time = 1.902 secs
FUNC = 0.999840
```

Versione ASSEMBLY base

3.2. Loop Vectorization:

Dopo la versione a singolo *float* si è passati all'utilizzo delle istruzioni vettorizzate del repertorio *SSE* (MOVAPS), ovvero implementando la tecnica *Loop Vectorization.* La seconda scelta progettuale è stata presa proprio durante l'adozione di questa tecnica, decidendo di utilizzare il **padding** per fare

in modo che i vettori delle dimensioni fossero multipli di 4. Così facendo si è semplificato il codice, non dovendo gestire cicli resto ed è stato possibile lavorare con dati allineati, vantaggio notevole in questo tipo di algoritmi; perché se il numero di dimensioni non fosse multiplo di 4, dato che ogni accesso alla matrice delle dimensioni dei pesci corrisponde a 4 accessi simultanei grazie alla *Loop Vectorization*, a partire dall'accesso della seconda riga della matrice si avrà sicuramente un accesso non allineato.

Le operazioni di padding sono state eseguite nel main, attraverso il metodo *void aggiungiResto(params* input, int resto)*, il quale provvede ad aggiungere ai vettori un numero di zeri tra 0 e 3, in relazione al risultato dell'operazione *4-d%4*. Grazie a questa operazione, è stato possibile adoperare i comandi MOVAPS invece che MOVUPS, sfruttando i benefici di operare con istruzioni allineate.

In figura è mostrato il codice in *ASSEMBLY* del metodo *func* con la *Loop Vectorization*. Si nota come le istruzioni rispetto al caso precedente utilizzano la "p" che sta per *packed* ovvero vettorizzate. Le istruzioni di *MOV* sono tutte allineate grazie al padding descritto in precedenza; inoltre alla fine del ciclo sono presenti delle *HADDPS* che sommano i 4 elementi presenti nel registro *XMM* a fine ciclo siccome durante il ciclo le operazioni sono vettorizzate (sommati 4 elementi per volta).

```
        mov     EAX, [EBP+input]
        mov     ECX, [EBP+xy]
        mov     ESI, [EAX + c]
        xorps   XMM0, XMM0
        xorps   XMM2, XMM2
        mov     EDX, 0
forjf:  imul    EDI, [EAX+d], dim
        imul    EDI, [EBP+i]
        add     EDI, EDX

        movaps  XMM1, [ECX + EDI]
        mulps   XMM1, XMM1
        addps   XMM0, XMM1
        movaps  XMM1, [ECX + EDI]
        mulps   XMM1, [ESI + EDX]
        addps   XMM2, XMM1

        add     EDX, dim*p
        mov     EDI, [EAX+d]
        imul    EBX, EDI, dim
        cmp     EDX, EBX
        jb      forjf

        haddps  XMM0, XMM0
        haddps  XMM0, XMM0

        haddps  XMM2, XMM2
        haddps  XMM2, XMM2

        mov     EBX, [EBP+indExp]
        movss   [EBX], XMM0

        subss   XMM0, XMM2
        mov     EAX, [EBP + indRis]
        movss   [EAX], XMM0
```

Metodo func32 ASSEMBLY versione con Loop Vectorization

In figura sono riportati i tempi di esecuzione della versione in *ASSEMBLY* base e con *Loop Vectorization*; si nota come la versione che adotta la tecnica di ottimizzazione *SIMD* sia più veloce.

```
FSS time = 1.902 secs
FUNC = 0.999840
```

Versione ASSEMBLY base

```
FSS time = 0.585 secs
FUNC = 0.999840
```

Versione ASSEMBLY Loop Vectorization

3.3. Loop Unrolling:

Un'ulteriore ottimizzazione è stata apportata con l'uso della *Loop Unrolling.* Lo scopo è quello di favorire il parallelismo implicito srotolando i cicli, ovvero inserendo nel ciclo non più una sola operazione, ma un certo numero di questa. Il ciclo, di conseguenza, itera incrementando l'indice del numero di operazioni svolte al suo interno; si è scelto come fattore di UNROLL 4. Con questa tecnica però, è divenuto necessario gestire i cicli resto. Di fatti le iterazioni, anche in conseguenza dell'uso della *Loop Vectorization*, lavorano su gruppi di 16 elementi (4*4, ovvero ognuna delle 4 chiamate lavora su vettori di 4 elementi), ma non vi è nessuna certezza che il numero di dimensioni o il numero di pesci sia multiplo di 16 (con il padding si è solo fatto in modo che le dimensioni fossero multiple di 4). Il fattore di UNROLL è stato impostato a 4, poiché con un fattore maggiore il numero di accessi in memoria sarebbe stato elevato a causa della mancanza di registri a 32 Bit, mentre con un numero inferiore in termini di efficienza non ci sarebbe stata una grande differenza con un ciclo senza *Unrolling.*

```
FSS time = 0.585 secs
FUNC = 0.999840
```

Versione ASSEMBLY Loop Vectorization

```
FSS time = 0.468 secs
FUNC = 0.999899
```

Versione ASSEMBLY Loop Vectorization + Loop Unrolling

Nella figura seguente è presente il codice della versione in *ASSEMBLY* con *Loop Vectorization* e *Loop Unrolling*. Nota subito all'occhio come il numero di istruzioni sia più elevato rispetto alla versione precedente, questo dovuto allo srotolamento del ciclo. Va evidenziato anche come in questo caso è presente un ciclo resto, siccome non si ha garanzia che il numero di dimensioni sia multiplo di 16.

```
                mov     EAX, [EBP+input]
                mov     ECX, [EBP+xy]
                mov     ESI, [EAX + c]
                xorps   XMM0, XMM0
                xorps   XMM7, XMM7
                mov     EDX, 0

forjf:          imul    EBX, [EAX+d], dim
                sub     EBX, dim*p*(UNROLL-1)
                cmp     EDX, EBX
                jge     forjRestf

                imul    EDI, [EAX+d], dim
                imul    EDI, [EBP+i]
                add     EDI, EDX
```

```
                movaps  XMM1, [ECX + EDI]
                movaps  XMM2, [ECX + EDI + p*dim]
                movaps  XMM3, [ECX + EDI + 2*p*dim]
                movaps  XMM4, [ECX + EDI + 3*p*dim]

                mulps   XMM1, XMM1
                mulps   XMM2, XMM2
                mulps   XMM3, XMM3
                mulps   XMM4, XMM4

                addps   XMM0, XMM1
                addps   XMM0, XMM2
                addps   XMM0, XMM3
                addps   XMM0, XMM4
                movaps  XMM1, [ECX + EDI]
                movaps  XMM2, [ECX + EDI + p*dim]
                movaps  XMM3, [ECX + EDI + 2*p*dim]
                movaps  XMM4, [ECX + EDI + 3*p*dim]

                mulps   XMM1, [ESI + EDX]
                mulps   XMM2, [ESI + EDX + p*dim]
                mulps   XMM3, [ESI + EDX + 2*p*dim]
                mulps   XMM4, [ESI + EDX + 3*p*dim]

                addps   XMM7, XMM1
                addps   XMM7, XMM2
                addps   XMM7, XMM3
                addps   XMM7, XMM4
                add     EDX, dim*p*UNROLL
                jmp     forjf

forjRestf:      mov     EDI, [EAX+d]
                imul    EBX, EDI, dim
                cmp     EDX, EBX
                jge     endForj

                imul    EDI, [EAX+d], dim
                imul    EDI, [EBP+i]
                add     EDI, EDX

                movaps  XMM1, [ECX + EDI]
                mulps   XMM1, XMM1
                addps   XMM0, XMM1
                movaps  XMM1, [ECX + EDI]
                mulps   XMM1, [ESI + EDX]
                addps   XMM7, XMM1
```

```
                add     EDX, dim*p
                jmp     forjRestf

endForj:        haddps  XMM0, XMM0
                haddps  XMM0, XMM0

                haddps  XMM7, XMM7
                haddps  XMM7, XMM7

                mov     EBX, [EBP+indExp]
                movss   [EBX], XMM0

                subss   XMM0, XMM7
                mov     EAX, [EBP + indRis]
                movss   [EAX], XMM0
```

Metodo func32 ASSEMBLY versione con Loop Vectorization e Loop Unrolling

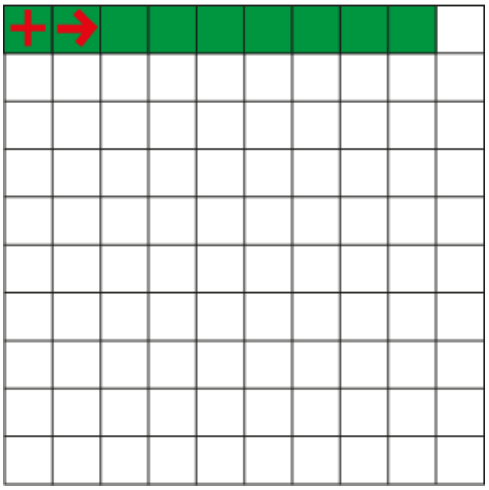
3.4. Cache Blocking:

Tra tutte le tecniche di ottimizzazione elencate in precedenza, l'unica che non è stata adottata è il *Cache Blocking*. La motivazione principale fa riferimento alla struttura del codice che è intrinsecamente *cache friendly*, poiché le istruzioni sono all'interno di cicli ordinati secondo la scelta della rappresentazione dei dati discussa nel primo capitolo. In tutti i casi infatti, le operazioni svolte tra più matrici non necessitano di analizzare più elementi, ma operano solo sull'elemento corrente. Ad esempio come si vede nel codice sottostante, sull'elemento corrente della matrice *deltaX[i, j]*, si assegna il valore della differenza tra la dimensione *y[d*i+j]* nuova e la dimensione *x[d*i+j]* vecchia; si nota come ci sia sempre una relazione tra la posizione della matrice a sinistra e a destra dell'assegnamento; inoltre ogni elemento delle matrici viene letto una sola volta, e dunque non è necessario racchiudere il ciclo in un blocco perché si sfrutta implicitamente l'utilizzo della cache.

```
for(i=0; i<n; i++){
        for(j=0; j<d; j++){
                deltaX[d*i+j] = y[d*i+j] - x[d*i+j];
                x[d*i+j] = y[d*i+j];
        }
}
```

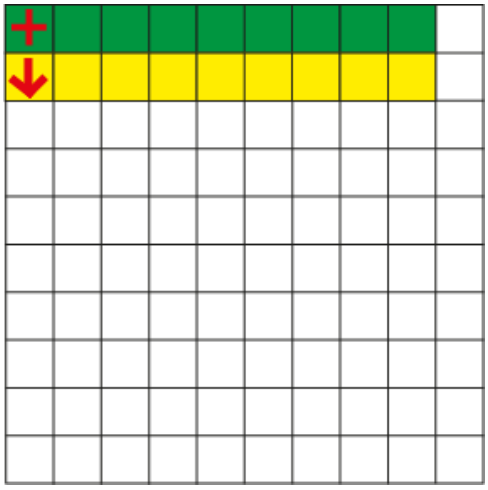
Per capire meglio cosa è stato detto in precedenza, ecco alcune figure che illustrano schematicamente i benefici di avere una memorizzazione dei dati secondo l'ordine di iterazione di questi ultimi dal punto di vista degli accessi in memoria aumentando il numero di *cache hit* e diminuendo quindi il numero di *cache miss*.

Le due immagini di seguito mostrano come vengono analizzate le matrici nell'approccio adottato. Ricordando che i dati sono memorizzati per riga, ad ogni iterazione se l'elemento corrente della matrice non è già presente in cache, questo verrà copiato in cache insieme alla riga in cui si trova o una sua parte se quest'ultima non entra interamente in cache. Poiché i cicli iterano proprio sulle righe, si avrà un numero di *cache miss* proporzionale al numero delle righe della matrice (se supponiamo che una riga della matrice entri interamente in cache), che è molto inferiore rispetto al numero di elementi totali della matrice.



Esempio iterazione for i – for j su matrice salvata per righe

Ciò non si sarebbe verificato se si fosse iterato in maniera contrario all'ordine di memorizzazione dei dati. Di seguito un esempio di iterazione per colonne e supponiamo che la matrice sia rappresentata in memoria come nel caso precedente, ovvero per righe. Si nota come il semplice passaggio da un elemento all'altro generi ogni volta un *cache miss*, poiché la prima volta si vuole accedere ad un elemento della prima riga, e dunque viene caricata in cache la prima riga; il secondo elemento da accedere si trova però nella seconda riga, e ciò genera ovviamente un *cache miss*: in numero proporzionale, dunque, alla dimensione della matrice.



Esempio iterazione for j – for i su matrice salvata per righe

Nel caso in cui ordine di iterazione e tipo di rappresentazione dei dati non siano concordi, il *Cache Blocking* sarebbe un'ottima soluzione per ridurre il numero di *cache miss*.

3.5. Open MP:

L'ultima ottimizzazione utilizzata fa riferimento alle tecniche di tipo MIMD. In particolare, sono state utilizzate le direttive Open MP per parallelizzare le chiamate ai metodi assembly usati all'interno dei cicli for. Per rendere il tutto più efficiente sono stati utilizzati i metodi assembly con *Loop Vectorization* e *Loop Unrolling*, opportunamente modificati affinché le operazioni venissero svolte solo su un ciclo e non su due. Ad esempio, il metodo assembly *movimentoIndividuale1* nelle versioni non Open MP iterava su due cicli: uno per il numero di pesci ed uno per le dimensioni. Per sfruttare al meglio i thread generati dalle direttive, movimentoIndividuale1 ha subito alcuni cambiamenti, ovvero il ciclo for che itera sui pesci è stato spostato in C, così da usare la notazione *#pragma omp parallel for*, mentre al suo interno è rimasta la porzione di codice che svolge le operazioni sulle dimensioni. Inoltre, per evitare la *Race Condition* è stato modificato il passaggio di alcuni parametri, ed si è fatta attenzione ad inserire la direttiva *OpenMP* solamente su cicli *for* che non generassero conflitti su variabili condivise. I metodi *ASSEMBLY* che hanno subito i cambiamenti per massimizzare l'efficienza grazie alle ottimizzazioni di tipo MIMD sono:

- *movimentoIndividuale1*
- *movimentoIndividuale2*
- *operatoreAlimentazione*
- *movimentoIstintivo*
- *movimentoVolitivo*

```
FSS time = 0,370 secs
FUNC = 0,999867
```

Versione ASSEMBLY Loop Vectorization + Loop Unrolling + OpenMP (32 bit)

Nella figura sottostante sono mostrate le differenze dei codici della porzione di ciclo *for* tra la versione senza e con *openMP*. Si vede come *indRandom* viene aggiornato solamente a fine ciclo, così da evitare problemi di *Race Condition*; conoscendo a priori il numero di iterazioni, *indRandom* viene aggiornato nel seguente modo: *indRandom += n*(d-zeri)*. Inoltre, il valore corretto di *indRandom* viene passato al thread *i* come **indRandom+i*(d-zeri)*; in questo modo il thread *i* scorre i valori del vettore dei random a partire dal parametro corretto del valore di *indRadom* passato. Da notare che anche per il metodo *movimentoVolitivo*, è stata effettuata una modifica simile, solamente che per quest'ultimo, *indRandom* viene aggiornato come *indRandom += n*, siccome in questo caso bisogna prelevare un numero random per ogni pesce e non per ogni dimensione di ogni pesce.

```
void movimentoIndividuale(params* input, MATRIX deltaX, VECTOR deltaF, int* indRandom, MATRIX y,int zeri){
        int i, j;
        int n = input->np;
        int d = input->d;
        MATRIX x = input->x;
        movimentoIndividuale1(input,indRandom,y,zeri);

  void movimentoIndividualeA(params* input, MATRIX deltaX, VECTOR deltaF, int* indRandom, MATRIX y, int zeri){
      int i, j;
      int n = input->np;
      int d = input->d;
      MATRIX x = input->x;
      int iR = *indRandom;

      #pragma omp parallel for
      for(i=0; i<n; i++){
          movimentoIndividuale1(input, *indRandom+i*(d-zeri), i, y, zeri);
      }
      *indRandom += n*(d-zeri);
```

Versione con OpenMP

4. Versione 64 bit/AVX:

Per l'implementazione della versione a 64 Bit è stato utilizzato il repertorio AVX. La versione assembly è stata sviluppata direttamente nella versione *Loop Vectorization* e *Loop Unrolling*. Le istruzioni, dunque, sono del tipo VMOVAPD, ovvero lavorano con vettori di *double* (8 byte), memorizzati nei registri del tipo *YMM* (256 bit). Rispetto alla versione 32/SSE il codice ha subito modifiche principalmente nella sintassi, mentre la logica alla base è rimasta quasi del tutto invariata, perché nonostante sia cambiata la dimensione dei registri dei valori, i registri vettoriali continuano a contenere solamente 4 elementi. La parte in linguaggio C è rimasta identica a quella usata per la versione a 32 bit, se non per il valore *type* settato a *double*. Le differenze più importanti per quanto concerne il codice *ASSEMBLY* riguardano il passaggio dei parametri (dato che con il repertorio AVX questi vengono passati direttamente in un insieme di registri), e operazioni come le HADDPS. Di seguito si possono notare alcune delle differenze tra le versioni 32/SSE e 64/AVX:

```
endForj:    haddps  XMM0, XMM0
            haddps  XMM0, XMM0

            haddps  XMM7, XMM7
            haddps  XMM7, XMM7

            mov     EBX, [EBP+indExp]
            movss   [EBX], XMM0

            subss   XMM0, XMM7
            mov     EAX, [EBP + indRis]
            movss   [EAX], XMM0
```

Codice ASSEMBLY 32/SSE

```
endForj:    vhaddpd         YMM0, YMM0, YMM0
            vperm2f128      YMM1, YMM0, YMM0, 00010001b
            vaddpd          YMM1, YMM0

            vhaddpd         YMM7, YMM7, YMM7
            vperm2f128      YMM2, YMM7, YMM7, 00010001b
            vaddpd          YMM2, YMM7

            vmovsd  [RCX], XMM1

            vsubsd  XMM1, XMM2
            vmovsd  [R8], XMM1
```

Codice ASSEMBLY 64/AVX

```
Coefficient file name: 'data/coeff64_256.ds2'
Random numbers file name: 'data/rand64_256_768_1250.ds2'
Initial fish position file name: 'data/x64_256.ds2'
Dimensions: 256
Number of fishes [np]: 768
Individual step [si]: 0.003000
Volitive step [sv]: 0.000400
Weight scale [w]: 0.040000
Number of iterations [it]: 1250
FSS time = 1.124 secs
FUNC = 0.999850

Done.
```

Esecuzione versione ASSEMBLY 64/AVX (256-768-1250)

4.1. Open MP

Anche per quanto riguarda la tecnica di ottimizzazione MIMD, le differenze con l'architettura a 32 Bit sono minime. I metodi *ASSEMBLY* infatti sono dichiarati allo stesso modo, il codice *C* rimane totalmente invariato, mentre le procedure *ASSEMBLY* fanno uso del repertorio *AVX* invece che il repertorio *SSE*. In figura è mostrato il valore del tempo si esecuzione dell'algoritmo con ottimizzazione *MIMD* nella versione a 64 Bit.

```
FSS time = 0,585 secs
FUNC = 0,999859
```

Versione ASSEMBLY Loop Vectorization + Loop Unrolling + OpenMP (64 bit)

5. Conclusioni:

Per concludere, dopo aver analizzato tutte le versioni realizzate, abbiamo notato come, la riduzione più importante in termini temporali sia data dalla traduzione del codice in alto livello in basso livello. In percentuale invece, si è ottenuta una riduzione di circa il 75% implementando l'ottimizzazione di tipo *SIMD*; una riduzione ulteriore di circa 20% implementando l'ottimizzazione di tipo ILP; infine una riduzione di circa 20-25% con l'uso dell'ottimizzazione di tipo *MIMD*. Bisogna considerare che tali valori non sono molto precisi; ciò è dovuto al fatto che i tempi di esecuzione sono inferiori ad 1 secondo, quindi molto dipendenti dallo stato corrente della macchina su cui si sta eseguendo l'algoritmo; per poter evidenziare un risultato con maggiore precisione, perciò, servirebbero tempi di esecuzione più elevati, ad esempio eseguendo l'algoritmo *fss* su dati avanti dimensione maggiore o su un maggior numero di iterazioni.

Il grafico sottostante rappresenta i tempi di esecuzione relativi alle due versioni richieste dalla traccia. Sulle ascisse sono presenti i vari livelli di ottimizzazioni implementate, mentre sulle ordinate sono presenti i tempi di esecuzione dell'algoritmo. La linea blu fa riferimento all'architettura a 32 Bit con repertorio *SSE*; la linea arancione fa riferimento all'architettura a 64 Bit con repertorio *AVX*. I tempi di ottimizzazione sono inversamente proporzionali al numero di ottimizzazioni implementate come si può vedere dall'andamento decrescente delle due linee.